DistribNet
(Draft)

Kevin Atkinson (kevin at atkinson dhs org)

December 1, 2003

Project Page: http://distribnet.sourceforge.net/
Mailing list:
http://lists.sourceforge.net/lists/listinfo/distribnet-devel

Abstract

DistribNet is a global peer-to-peer Internet file system in which anyone can tap into or add content to. This paper gives an overview of DistribNet.

1 Overview

DistribNet is a peer-to-peer file system in which anyone can access or add content to. A key feature of DistribNet is that it does not require access to a central server in order to contribute content. A user simply adds a file to the network and it will be available for others to download. Furthermore once I file is uploaded to DistribNet it will always be available unless the file is determined to be completely worthless. Thus a DistribNet file can be cited with confidence that anyone wishing to access the document will be able to download it, which is very different from the situation of citing a document on the WWW.

2 DistribNet Architecture

DistribNet will have several distinct types of keys. The behavior of the key depends on several parameters:

Routed or non-Routed Routed data will essentially be stored based on a distributed hashing algorithm so it can always be found. Non-routed data will be stored where it will be the most beneficial performance and availability wise. Routed data will typically be small (under 1k) and will often point to non-routed data.

Map or Data Data in DistribNet will either be indexed based on there identification or there content. Data index via there identification are *Map* keys while data indexed on there content are *Data* keys. Map keys are generally small and are primarily used to point to other data. Since they are small they will always be routed. Map keys can also be updated. Data keys, on the other hand can be of arbitrary size and are not routed. Since data keys are indexed based on the actual content they can not be updated.

Permanent or Cached Data in DistribNet is stored where it will do the most beneficial both performance and availability wise. Depending on the location of the data it can either be considered a Permanent key or a Cached key. Cached keys are freely deleted when space is needed while permanent keys are only deleted if the data is also available somewhere else on the network. A cached key can become a permanent key if necessary to insure that data does not disappear from the network.

3 DistribNet Routing

The routing algorithm for DistribNet was originally based on Pastry[4], but several key modifications were made. It is now probably closer to Kademlia[5] than Pastry, however it still maintains several elements of Pastry. This section will assumes the reader is family with Pastry and will focus on how it differers from such.

Each node on DistribNet is uniquely identifies by the 160-bit SHA-1 hash of the public key. Since SHA-1 hashes are used the nodes will be evenly distributed. (Note: For efficiency reasons I am strongly considering only using the first 64 bits of the key and not allow nodes with duplicate keys to join the network.) Keys in DistribNet are stored based solely the XOR metric in the same manor as is done in Kademlia. In the XOR metric distance is determined by taking the exclusive or of the keys and then treating the resulting value of an integer in big endian format. For example the keys 0xF0 and 0x1F will have a distance of 0xDF. The XOR metric is used from start to finish, unlike with Pastry which switches to using numerical distance for the last hop.

The routing table in DistribNet contains 8 rows with each row containing 2^4 entries each. In general, DistribNet tries to maintain at least two nodes for each entry. The number of rows does not need to be fixed and it can change based on the network size. It may also be possible that the number of entries per row does not necessarily have to be fixed. However, This idea has not been exported in depth. Four was chosen as the base size for several reasons 1) it is a power of two, 2) when keys are thought of as a sequence of digits a base size of 4 means that the digits will be hexadecimal, 3) the Pastry paper hinted that 4 would be a good choice. The number of rows was chosen to be large enough so that there is no possibility that the last row will be used when dealing with a moderate size network during testing.

Unlike Pastry their is no real Leaf set. Instead the "leaf set" consists of all rows which are not "full". A full row is a row which contains 15 full entries with am extra empty entry being the one which represents the common digit with the node's key, and thus will never be used. Not having a true "leaf set" simplifies the implementation since a separate list does not need to be maintained and the routing algorithm remains the same instead of switching to numerical distance as Pastry does. This also means that all the nodes in the leaf set will maintain the same set. I have not determined if this is a good or bad thing. It also has not been determined if maintaining any sort of leaf set at all is necessary, as Kademlia does not. I, however, believe that maintaining a leaf set will lead to a more robust network that is less sensitive to network failures.

A row is considered full in DistribNet if 15 of the 16 entries are full in the current node AND other nodes on that row also have 15 of the 16 entries full. For each full row DistribNet will try to maintain at least two nodes for each entry. This way if one node goes down the other one can be used without effecting performance. When a node is determined to be down (as oppose to being momentary offline) DistribNet will try to replace it with another node that is up. With this arraignment is is extremely likely that at least one on the two nodes will be available. A full row can become a leaf row if the entry count drops below 15.

For each non-full row (ie in the Leaf Set) DistribNet will attempt to maintain as many nodes as are available for that entry so that every other node in the leaf set is accounted for. From time to time DistribNet will contact another node in the leaf set and synchronize its leaf set with it. This is possible because all nodes in the leaf set will have the same set. Down nodes in the leaf set will be removed, but the criteria for when a node is down for a leaf set is stricter than the criteria for a full row. If a leaf row becomes a full row than excess nodes will be removed.

DistribNet also maintains an accurate estimate on the number of nodes that are on the network. This is possible because unlike with network such as freenet, all

nodes are accounted for.

4 Mutable Keys

Mutable keys in DistribNet net are keys which are indexed based on there identification are will generally point to other data. Furthermore, since all map keys are small they will always be routed. DistribNet will have several different types of map keys: Map Keys, Pointer keys, and Appendable Lists.

4.1 Map Keys

Map keys are mutable keys which point to data keys. Map keys will contain the following information:

- Short Description
- Public Namespace Key
- Timestamped Index pointers
- Timestamped Data pointers

At any given point in time each map key will only be associated with one index pointer and one data pointer. Map keys can be updated by appending a new index or data pointer to the existing list. By default, when a map key is queried only the most recent pointer will be returned. However, older pointers are still there and may be retrieved by specifying a specific date. Thus, map keys may be updated, but information is never lost or overwritten.

4.2 Pointer Keys

Pointer keys are keys which point to the location of non-routed keys. They are essential to finding real data on the network. Pointer keys, like all routed keys, will be distributed based on the numeric distance of the hash of the key from the hash of the node's identification. Since pointer keys contain very little data they will be an extremely large amount of redundancy. Pointer keys will contain two types of pointers. Pointers to permanent keys and pointers to cached keys. Pointer keys on different nodes will all contain the same permanent pointers but will only contain pointers to cached keys to nodes which or near by. There will be an upper limit to the number of pointers within an pointer key any one node will have.

4.3 Appendable Lists

Appendable list are map keys in which, as their name suggest, can only be appended to. They will behavior similar to Map keys but each entry can contain any arbitrary value rather than just pointers to data keys. Each entry in a appendable list is timestamped and it will be possible to only retrieve a subset of the list based on the timestamps.

5 Cache Consistency

Maintaining cache consistency is a difficult problem for any network. Some networks, such as Freenet, avoid the issue all together by not having mutable keys. Other networks only keep cached copied of data around for a short time span, therefor reducing the chance of the cache data being out of date. Once a cached copy gets out of date the node either throws the copy away, or checks to see if a newer copy is available. Either approach creates unnecessary network traffic. Another approach is to have the server notify other nodes when ever the data changes. This approach will not scale well as the number of nodes a server needs to keep track of will grow with the network and is completely impractical for a large distributed network. A similar, but more scalable, approach is for nodes on the network to notify each other when ever a key changes. This is the approach that DistribNet uses.

The other issue that needs to be dealt with when is conflicts. That is when two different people modify the same key at nearly the same time. DistribNet avoids this problem by only allowing keys to be added to.

6 Data Keys

Data keys is DistribNet are keys which are indexed based on there content and hold an arbitrary amount of non-mutable data. Since they are large they will generally not be routed.

6.1 Details

Data keys will be stored in maximum size blocks of just under 32K. If an object is larger than 32K it will be broken down into smaller size chunks and an index block,

Data Block Size:	2^15 - 128 = 32640
Index block header size:	40
Maximum number of keys per index block:	1630
Key Size:	20

Maximum object sizes:

```
direct   => 2^14.99 bytes , about 31.9 kilo
1 level  => 2^25.66 bytes , about 50.7 megs
2 levels => 2^36.34 bytes , about 80.8 gigs
3 levels => 2^47.01 bytes , about 129 tera
4 levels => 2^57.68 bytes
5 levels => 2^68.35 bytes
```

also with a maximum size of about 32K, will be created so that the final object can be reassembled. If an object is too big to be indexed by one index block the index blocks themselves will be split up. This can be done as many times as necessary therefore providing the ability to store files of arbitrary size. DistribNet will use 64 bit integers to store the file size therefore supporting file sizes up to 2^64-1 bytes.

Data keys will be retrieved by blocks rather than all at once. When a client first requests a data key that is too large to fit in a block an index block will be returned. It is then up the client to figure out how to retrieve the individual blocks.

Please note that even though that blocks are retrieved individually they are not treated as truly independent keys by the nodes. For example a node can be asked which blocks it has based on a given index block rather than having to ask for each and every data block. Also, nodes maintain persistent connections so that blocks can be retrieved one after another without having to re-establish to connection each time.

Data and index blocks will be indexed based on the SHA-1 hash of there contents. The exact numbers of as follows:

Why 32640? A block size of just under 32K was chosen because I wanted a size which will allow most text files to fit in one block, most other files with one level of indexing, and just about anything anybody would think of transferring on a public network in two levels and 32K worked out perfectly. Also, files around 32K are rather rare therefor preventing a lot of of unnecessary splitting of files that don't quite make it. 32640 rather than exactly 32K was chosen to allow some additional

information to be transfered with the block without pushing the total size over 32K. 32640 can also be stored nicely in a 16 bit integer without having to worry if its signed or unsigned.

However, the exact block size is not fixed in stone. If, at a latter date, a different block size is deemed to be more appropriate than this number can be changed.

6.2 Storage

Permanent data keys will be distributed essentially randomly. However, to insure availability the network will insure try to insure at least N nodes contain the data. Keys that are more variable will be Nodes which are responsible for maintaining a permanent key will know about all the other nodes on the network with are also responsible for that key. From time to time it will check up on the other nodes to make sure they are still live and if less than N-1 other nodes are live it will pick another node to to ask to maintain a copy of the key. It will first try nodes which already have the key in its cache and if they all refuse or none of them do. It will chose a random node to ask and will keep trying until some node accepts or one the original nodes becomes live again. The exact value for N and how hard DistribNet tries to insure a keys availability will be based on its estimated worth.

Cached data keys will be DistribNet based on where it will do the most good performance wise. For the initial implementation cached keys are simply stored on the node that downloaded them. When space ones short the cached keys the oldest keys will be deleted. Age is based on the last time a user of the local node accessed the file. Age does not include accesses from other nodes. The theory here is that if another node accessed a key it will also store a copy of the key, thus there is no need to cache it.

6.3 Retrieval

When a node A wants to retrieve a key K either two things will happen. If it has good reason to believe that a nearby node has the key it will attempt to retrieve it from that node, otherwise it will send a request to find other nodes which have the key.

To do this, node A will contact a node, B, whose key is closer to K than node's A key. Node B which will in tern contact C etc, until an answer is found which for the sake of argument will be node E. Node E will then send a list of possible nodes L which contain key K directly to node A. Node E will then send the result to node D,

which will send it to C, etc. Node E will also add node A to list L with probability of say 10%, Node D will do the same but with a probability of say 25%, etc. This will avoid the problem having the list L becomes extremely large for popular data but allow nodes close to A to discover that A has the data since nodes close to A will likely contact the same nodes that A tried. Since A requested the location of key K it is assumed that K will will likely download the data. If this assumption is false than node A will simply be removed at the list latter on.

Once A retrieves the list it will pick a node from the list L based on some evaluation function, lets say it picks node X. Node X will then return the key to node A. The evaluation function will take several factors, into account, including distance, download speed, past reputation, and if node A even knows anything about the new node.

If node X does not send the key back to node A for what ever reason it will remove node X from the list and try again. It will also send this information to node B so it can consider removing node X from its list, it will then in term notify node C of the information, etc. If the key is an index block it will also send information of what parts of the complete key node X has. If the key is not an index block than node a is done.

If the key is an index block than node A will start downloading the sub-blocks of key K that node X has. At the same time, if the key is large or node X does not contain all the sub-blocks of K, node X will chose another node from the list to contact, and possible other nodes depending on the size of the file. It will then download other parts of the file from the other nodes. Which blocks are download from which nodes will chance based on the download speed of the nodes so that more blocks are download from faster nodes and less from slower, thus allowing the data to be transfered in the least amount of time. If after contacting a certain number of nodes there are still parts of the key that are not available on any of those nodes, node A will perform a separate query for the individual blocks. However, I image, in practice this will rarely be necessary.

7 Distance determination

One very course estimate for node distance would be to use the XOR distance between two nodes ip address since closer nodes are likely to share the same gateways and nodes really close are likely to be on the same subnet. This is what the current implementation of DistribNet does.

Another way to estimate node distance releases on the the fact that node distance,

for the most part, obeys the triangle inequality. For each node in the list of candidate nodes some information about the estimated distance between that node, node E, in the list and the node storing the list is maintained by some means. For node A to estimate the distance between a node on the list, node X, and itself all it has to do is combine the distance between it and E with the distance between E and X. The combination function will depend on the aspect of distance that is being measured. For the number of hops it will simply add them, for download speed it will take the maximum, etc.

8 Determining Worth

One important aspect of DistribNet is to determine how valuable a key is and thus how hard the network should try to keep it available. Worth should not be based on how popular a file is a the moment but how popular a file is over the long term. Furthermore a file which is popular over a long period of time should be given more value than a file which is really popular but only for a short period of time. Worth should be based on a combination of long term access patterns as well as what users think of the file. Authors should also be able to influence the worth of a document, for example a draft of a paper should have less worth than the published paper.

In order to determine worth based on usage patterns access to a file must be logged in some way. Unfortunately logging every single access of a file from the beginning of time is not very practical so it needs to be summarized in some manner than will accurately reflect overall usage patterns. Simply maintaining a count of the number of times a file is accessed is not sufficient as that will not be able to distinguish files which are popular over a long period over file which are really popular but only for a short time. It also needs to be possible to merge the access logs of different nodes or at least combine there worth rankings in some intelligent manner. Needless to say I have not found a good solution to this problem. However, I hope to have a solution to this problem in the initial release of DistribNet.

In order to determine worth based on users opinion of a file there needs to be a way for users to vote on a file. Furthermore, there needs to be a way to prevent a user from voting multiple times and thus artificially increasing the worth of a file. Future versions of DistribNet may support this, but I have no plans on implemented it now.

Authors should also have a say on the worth of a file. However, there should only be able to influence so much. Naturally they should not be able to give a file a

high worth value, but they should also not be able to give it too low of a value. For example an author should not be able to release a document which becomes very popular and then some time latter deem it worthless and thus remove it from the network. In DistribNet I plan to allow a author to mark a document as deleted, but won't allow them to actually purge it from the network. instead deleting a document will lower it worth value by a small percentage. If no one was interested in the file in the first place than this small decrees in worth will likely cause the file to "fall off" the network fairly quickly. However if lots of people are still accessing a file it will still be available.

9 Limitations

Because they is no indirection when retrieving data most of the data on any particular node would be data that a local node user requested at some point in time. This means that it is fairly easy to tell what which keys a particular user requested. Although complete anonymity for the browser is one of my anti-goals this is going a bit to far. One solution for this is to do something similar that GNUNet does which is described in [3].

It is also blatantly obvious which nodes have which keys. Although I do not see this as a major problem, especially if a solution for the first problem is found, it is something to consider. I will be more than happy to entertain solutions to this problem, provided that it doesn't effect effectually that much.

10 Implementation Details

An implementation for DistribNet is available at http://distribnet.sourceforge.net/. Unless otherwise stated everything described in this paper is implemented in my private copy of DistribNet. The latest public version, however may be a bit older, and thus not have every aspect implemented.

10.1 Physical Storage

Blocks are currently stored in one of three ways

1. block smaller than a fixed threshold (currently 1k) are stored using Berkeley DB (version 3.3 or better).

2. blocks larger than the threshold are stored as files. The primary reason for doing this is to avoid limiting the size of data store by the maximum size of a file which is often 2 or 4 GB on most 32-bit systems.

3. blocks are not stored at all, instead they are linked to an external file out side of the data store much like a symbolic link links to file out side of the current directory. However since blocks often only represent part of the file the offset is also stored as part of the link. These links are stored in the same database that small blocks are stored in. Since the external file can easily be changed by the user, the SHA-1 hashes will be recomputed when the file modification data changes. If the SHA-1 hash of the block differs all the links to the file will be thrown out and the file will be relinked. (This part is not implemented yet).

Most of the code for the data keys can be found in data_key.cpp

10.2 Determining the amount of space used

When determine the amount of space used only large blocks are considered. That is only blocks which are stored as actual files will be counted. This is because predicting the amount of space a key will take to store in the database is not straight forward. It is easy to find out the current space used by a database file but it is not easy to determine what to do to decrease the size due to the large amount of meta-data stored in a database. With large blocks it is fairly safe to assume that the amount of space used is approximately the size of the file, the size of the metadata is insignificant due to the relatively large size of the file. Thus to free a N amount of space simply keep deleting files until the sum of size of the deleted files is larger than N.

Of course, the amount of size the database used is significant and the amount of data stored in it should be limited. I am just not sure how to do that. The best solution may be to simply limited the number of entries stored in the database.

10.3 Language

DistribNet is/will be written in fairly modern C++. It will use several external libraries however it will not use any C++ specific libraries. In particular I have no plan to use any sort of Abstraction library for POSIX functionally. Instead thin wrapper classes will be used which I have complete control over and will

serve mainly to make the process of using POSIX functions less tedious rather than abstract away the details of using them.

11 Freenet

DistribNet will be like Freenet in many ways and aims to solve many of the same problems of Freenet. However, the focus of DistribNet is completely different from Freenet. DistribNet main concern is about performance and simplicity while Freenet main concern is about anonymity and security. DistribNet will defiantly have some security but it is just not its primary focus. Also with Freenet the availability of a document depends only on its current popularity. Non-popular documents are very likely to disappear forever from the Freenet network, making it unsuitable for long term storage.

12 BitTorrent

Shortly after starting DistribNet BitTorrent become available. BitTorrent solves one specific problem that DistribNet also aims to solve. It solves the problem of overloading of sharing the network load in order to avoid having to provide lots of bandwidth on a central server. And it solves that problem very well. In fact in the short time since its initial release it became the defacta way to distribute large files such as Linux distributions and Fansubbed anaime.

However, that is the only problem it solves. In particular it still requires a central sever in order to distribute files. It is just that the central server does not have to provide a lot of bandwidth. The file is only available as long as the central server is hosting the file. Furthermore, if for some reason the central server goes down the file will no longer be available. Thus it does not really solve the problem of long term availability of a file. Just as a web site can disappear or move so can a BitTorrent file.

Nevertheless it does solve the bandwidth problem very well and is defiantly worth in investigating as some of its methods could also be used by DistribNet to help DistribNet solve the bandwidth problem as well or better than BitTorrent does. When it was first released the author did not write very much about how it works thus making it difficult to understand. However, there is now a paper describing BitTorrent which I at very least plan to look at before I consider the initial DistribNet Implementation.

References

[1] GNUNet. http://www.ovmj.org/GNUnet/ and http://www.gnu.org/software/GNUnet/

[2] Freenet. http://freenet.sourceforge.net/

[3] Krista Bennett and Christian Grothoff. “GNUnet – anonymity for free”. http://gecko.cs.purdue.edu/GNUnet/papers.php3

[4] Antony Rowstron and Peter Druschel. “Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems”. http://research.microsoft.com/~antr/Pastry/pubs.htm

[5] Kademlia: XOR metric-based routing. http://kademlia.scs.cs.nyu.edu/

[6] BitTorrent http://bitconjurer.org/BitTorrent/